Benchmarking Memory Performance with the Data Cube Operator

Michael A. Frumkin, Leonid V. Shabanov
NASA Advanced Supercomputing (NAS) Division
NASA Ames Research Center, Moffett Field, CA 94035-1000
frumkin@nas.nasa.gov
CrossZ Solutions USA,
leonid.shabanov@crossz.com

March 30, 2004

Abstract

Data movement across a computer memory hierarchy and across computational grids is known to be a limiting factor for applications processing large data sets. We use the Data Cube Operator on an Arithmetic Data Set, called ADC, to benchmark capabilities of computers and of computational grids to handle large distributed data sets. We present a prototype implementation of a parallel algorithm for computation of the operator. The algorithm follows a known approach for computing views from the smallest parent. The ADC stresses all levels of grid memory and storage by producing some of 2^d views of an Arithmetic Data Set of d-tuples described by a small number of integers. We control data intensity of the ADC by selecting the tuple parameters, the sizes of the views, and the number of realized views. Benchmarking results of memory performance of a number of computer architectures and of a small computational grid are presented.

1 Introduction

1.1 Memory Performance and Data Intensive Applications

Memory hierarchy of modern machines is growing in many directions: in size, in depth, and in complexity [11, Ch. 5]. Some computers employ dedicated multilevel caches (SGI Origin and Altix), others employ shared multilevel caches (IBM Power4), or use a combination of caches with vector registers (Cray X1), or unconventional architectures to hide memory latency (Cray MTA and Stanford STREAM processor). In spite of these efforts, even the best implementations of many important scientific codes on cache based machines achieve only 10-20% of peak machine performance due to slowness in feeding data to processors. For data intensive applications, performing a few operations per datum and accessing data in a random fashion, memory performance is the critical factor.

Two commonly used memory performance measures, bandwidth and latency, can be applied only for extreme cases of applications where all memory accesses are well vectorized or each access is an L2 miss. A memory performance measure which can be used to estimate performance of data intensive applications should reflect performance of all relevant memory components (from L1 to I/O). Several benchmarks are available for evaluation of memory and I/O systems, including STREAM and PTRANS [13], HINT [12], the recently developed NAS BTIO [30], and TPC transaction processing benchmarks [11], Ch. 7.9. The STREAM and PTRANS benchmarks measure memory bandwidth by accessing contiguous memory locations and sending data to a processor (STREAM) or between processors (PTRANS). The HINT benchmark computes $\int_0^1 \frac{1-x}{1+x} = 2\ln 2 - 1$ using a hierarchical integration method [12]. As precision of the computation increases, the hierarchical integration uses a finer partition of the interval which increases date set size. A drop in efficiency of the computation indicates that the data set does not fit in cache. A similar probing of the memory caches can be accomplished by accessing the memory with a fixed stride [11, p. 513]. NAS BTIO benchmark is designed to test the capability of systems to support parallel I/O. TPC benchmarks are designed to compare performance of query systems rather than to benchmark memory or I/O performance.

With computational grids coming on-line such as NASA's Grid Test Bed, TeraGrid, UNICORE and others, data sharing among research groups becomes a real-time activity. Distributed data access and processing become an essential part of many scientific and commercial applications and is supported by grid middleware such as Globus Toolkit, Load Sharing Facility (LSF), Storage Resource Broker (SRB), and Akamai data caching technology. The grid middleware has a major impact on a grid user's ability to access vast storage resources of computational grids. Currently, neither do the users have a tool to evaluate this middleware nor do developers have a commonly accepted measure of efficiency of their middleware. Most computer memory benchmarks are confined to a single machine and can not easily be extended to grids. The four NAS Grid Benchmarks ED, HC, MB, and VP are coarse grained and computationally intensive (see a taxonomy of grid applications in [25]) and cannot easily be transformed into a meaningful data intensive grid benchmark. A number of NASA's applications, such as satellite data assimilation, combining results of large scale distributed simulations, and transactions in distributed databases, are proprietary, too specific, and too complex to serve as a benchmark.

We propose a data intensive application benchmark which generates a large volume of data and, depending on the input data size, can be used to benchmark performance of any level of computer memory, from L1 to I/O and distributed storage of computational grids. By varying the size of input data, the benchmark can spill data across a few top levels of the computer memory hierarchy, making it also a good tool for obtaing a memory signature. This new benchmark, *Data Cube* (DC), takes a synthetic data set described by a small number of parameters and generates multiple views of this set. Informally, it can be classified as multidimensional sorting. Multiple processors can work in parallel to measure combined performance of multiple I/O systems attached to a machine. Futhermore, the parameters of the input data set can be chosen to saturate I/O systems of the largest existing machines, so that multiple grid hosts may be efficiently used to reduce the benchmark turn-around time. This property allows us to test grid middleware which finds appropriate grid resources and assigns benchmark subtasks to the various grid resources.

The DC benchmark performs a data intensive operation known in data mining as the *Data Cube Operator* (DCO). Informally, DCO computes views of a (maybe distributed) data set represented as a set of tuples. For a chosen set of attributes, a view is a sorted set of the tuples with attributes from the set. To generate a view, DCO performs $O(\log\ n)$ memory accesses per tuple, where n is the number of tuples. A view can be generated either from the original data set or from a *parent* (a view having one more attribute than the target view). This property allows us to split DCO into tasks having small intertask communications and to distribute the tasks across processors or/and grid hosts. A natural measure of the DCO performance is **TU**ples generated **P**er **S**econd (TUPS). TUPS represents the rate at which DCO generates tuples.

1.2 The Data Cube Operator

The main subject of data warehousing, On-Line Analytic Processing (OLAP), decision support database systems, data mining systems, and resource brokers, is a data set represented as a list of tuples. A tuple t of a data set having d dimension attributes and a single measure attribute can be represented as $t = (i_1, \ldots, i_d, c)$, where each dimension attribute i_j assumes values in an interval $[1, m_j - 1]$, and c is a cost function (a measure) associated with $(i_1, \ldots, i_d)$. The goal of OLAP is to assist users to discover patterns and anomalies in the data set by providing short query execution times [24].

A standard tool of OLAP is the DCO [9] which computes views (or group-bys) of a data set. For a chosen subset of k attributes, a view is a sorted set of k-tuples containing only the chosen attributes with accumulated measures of the duplicates. DCO computes views of interesting subsets of the dimensions. For example in [3, 14] there are proposed approaches for mining multi-dimensional association rules and answering iceberg queries by computing an iceberg cube containing views exceeding a certain threshold.

The input data sets and some of the materialized views usually do not fit in core memory, thus DCO computation requires a careful reuse of data loaded into the main memory (and all levels of cache). Computations of the DCO feature intensive data traffic across various levels of memory, making DCO especially interesting as a data intensive benchmark. Also, the size of the DCO output is usually significantly larger than the size of the input. Many papers are devoted to efficient computation of the DCO [15, 17, 23, 31] including parallel DCO computation algorithms [5, 18, 21]. To improve the efficiency of querying data cubes, a number of publications consider calculation and storage of data cubes as condensed cubes [29] or as other highly compressed structures [26].

For the reference implementation, we choose a greedy algorithm [15] that computes each view from a smallest parent. We assume that all attribute values are integers. Although real OLAP data sets and existing OLAP benchmarks [22, 28] use mostly strings as attribute values, this is not a significant limitation, since strings can be enumerated by integers (using hashing, for example). One of the advantages of using integer attribute values is reduction in the size of the input data sets and of the materialized views.

2 The DC Benchmark

2.1 Features and Parameters

There exist data sets to test OLAP systems, DCO algorithms, and data mining algorithms, for example, the ABP-1 and TPC-C,H,R,W benchmark databases [11, 22, 28]. For benchmarking purposes, the most appropriate is a synthetic parametrized data set which can be generated by a small program. This would make the data set scalable, the distribution of the benchmark manageable, and verification simple. Also, a synthetic data set, as in many real applications, can be generated in a distributed fashion, saving the effort and the overhead of splitting and distributing the data set on a computational grid.

In available synthetic data sets, the tuples are randomly generated. These data sets do not provide any means to control the sizes of the views. One can estimate the view sizes using sampling or some analytical methods [15, 26]. In [7] we introduced *the Arithmetic Data Set* (ADS), which is generated by a random number generator but has the advantage of a priori known sizes of the views.

ADS S is a subset of a group Q defined by

$$Q = \bigoplus_{i=1}^{d} (\mathbb{Z}/m_i\mathbb{Z})^*,$$

where $(\mathbb{Z}/m_i\mathbb{Z})^*$ is the set of integers modulo m_i relatively prime with m_i. An element of S can be represented by a tuple $x = (x_1, \ldots, x_d)$, where x_i is a modulo m_i residue. The subset S is defined by a seed $s = (s_1, \ldots, s_d) \in Q$, a generator $g = (g_1, \ldots, g_d) \in Q$, $s_i, g_i \neq 0,\ i = 1, \ldots, d$, and the total number of elements n:

$$S = \bigcup_{j=0}^{n-1} (s_1 g_1^j, \ldots, s_d g_d^j),$$

where the multiplication operations are within group Q. For any subset of k different cube dimensions $I = \{i_1, \ldots, i_k\} \subset \{1, \ldots, d\}$, the I-view of $x \in Q$ is defined as a projection of x on the I-face of the cube:

$$x_I = (x_{i_1}, \ldots, x_{i_k}).$$

The I-view of S is the set of I-views of all elements of S, or $S_I = \{x_I | x \in S\}$. If q_i is the smallest integer such that $g_i^{q_i} = 1 \mod (m_i)$, for the number of distinct elements in S_I, we have a formula $|S_I| = min(n, \mathrm{LCM}_{i \in I}(q_i))$[1], [7].

2.2 Choice of the Measures

In real applications, the sum of measures of all tuples in the view is used to characterize a view. In the benchmark, we use a single checksum for testing correctness and completeness of the computations. For this purpose, it is important that

- the measure of a tuple can be computed independently of other tuples
- the measure of a view cannot be calculated unless all tuples of a view have been generated
- the checksum is a separable function of the checksums of the views

[1] LCM stands for the Least Common Multiple.

To meet these requirements, we limit the maximum measure value by an arbitrarily chosen number $M = 31415$. Then we define the measure of a tuple $x = (x_1, \ldots, x_d)$ to be

$$\mu(x) = X * g \mod M$$

where X is the maximum of the attribute values of x and g is the first seed of x. Finally we define the checksum of the view $I = \{i_1, \ldots, i_k\} \subset \{1, \ldots, d\}$ to be

$$c(I) = \sum_{x \in S_I} (\nu(x) * \mu(x) \mod M)$$

where $\nu(x)$ is the sequence number of a tuple in S_I. As a result, the checksum of a view does not exceed M times the total number of the input tuples. Finally, the checksum of the benchmark is a sum of checksums of all generated views.

3 Implementation of Data Cube Computation

3.1 Approaches to Data Cube Computation

Since the publication of [9], a number of sequential and parallel data cube computation algorithms have been developed. These algorithms constitute two main groups, depending on whether they compute the views by means of sorting or by a hash table [8]. Each group employs similar optimizations: smallest-parent, cache-results, amortize-scans, [23, 15]. A share-sort optimization is specific to the sorting based algorithms. The *ArrayCube* [31] algorithm, based on *Multi-Way Array-Based* method, is another class of DCO computation algorithms. It uses a chunk-offset compression technique to deal with sparse data and memory management and performs a pipelined tuples aggregation. *ArrayCube* is the first practical algorithm designed for multidimensional OLAP systems. The generated data cubes often are stored as condensed or highly compressed cubes [29, 26, 17] to improve the efficiency of querying.

The views can be computed either in top-down or bottom-up manner [23, 3]. Such algorithms as *PipeSort*, *PipeHash*, and *Overlap* use the top-down approach. The *PipeSort* algorithm actually determines the sequence of views by finding a minimum weight matching in a bipartite graph. If the view size decreases as a function of the number of view attributes, these algorithms outperform many other algorithms.

Many DCO algorithms [9, 15, 23] use a smallest-in-size view "parent" from a set of already calculated views to create a new view. For certain classes of aggregation functions, dependency among related views can be represented as a search lattice [15]. The optimal sequence of views can be determined by solving a minimum spanning tree problem where cost of each node is the view size.

3.2 The Top-Down Data Cube Computation

For the reference implementation, we choose the top-down, sort-based data cube computation which uses the smallest-parent heuristic. The algorithm reads ADC data tuple-by-tuple from a file. It inserts a tuple into a balanced tree using dimension attributes as a key. If a tuple with the key is found in the tree, the measure attribute values are aggregated. If a view fits into main memory, the algorithm performs all aggregations 'on-the-fly'. After a view is built, the next view is computed from a smallest parent view. The algorithm proceeds until the computation of all views has been completed.

We use balanced trees (namely, *Red-Black*-trees) to aggregate data in a sort-base algorithm because they provide a simple way to aggregate data on-the-fly. This leads to a simple implementation of both internal and external branches of DCO. For data with a small number of duplicates, a regular sort will likely outperform the balanced trees. However, if the collapsing ratio is moderate, this performance gain will not be significant. Other balanced trees (for example, *AVL*-trees or some *B*-tree modifications) are suitable to aggregate data in main memory. However, experiments show that the performance gain is not significant relative to the *RB*-trees. Potentially, one can use variants of *B*-trees (for instance, *B*+-trees or *B**-trees) to aggregate data in an external case. That would be convenient because no merging would be necessary, but extremely inefficient. Despite improvements in cache performance, the *B*+-trees show bad performance for large random data sets. If, for example, a *B*+-tree cache can store only 10-20% of data, the high chance of reading at least one page from disk in any insert operation results in a large I/O volume. To be competitive, an external sort with multi-way merging has to significantly outperform any *B*-tree implementations.

Our sequential algorithm performs a dynamic task planning. It computes the data cube in a top-down, level-by-level manner. The algorithm starts to compute views with the given number of attributes when all views of the larger number of attributes are completed. To compute a new view, the algorithm chooses a smallest ready parent. The algorithm uses simple data structures to dynamically maintain the search lattice (a weighted graph).

If a view does not fit into main memory, the algorithm uses an external sorting. In this case, the algorithm uses balanced trees to form sorted chunks of the view. Each chunk contains only distinct tuples. Finally, the view is assembled from the chunks by means of multi-way merging.

3.3 DC Performance Model

Most DC execution time is spent on accessing data: fetching data from memory/disk, sorting the data, and writing the data to memory/disk. The main DC operation is the insertion of a tuple into the *RB*-tree. Some additional operations include balancing of the tree, bookkeeping operations, reading/writing internal buffers, and constant time operations, such as memory allocations.

For a view containing d_i attributes, the size of each node of an *RB*-tree has $u_0 d_i + u_1$ bytes, where u_0 and u_1 are constants[2]. Since we are using balanced trees, the number of nodes from the root to a leaf is between $\log n$ and $2 \log n$ for an *RB*-tree, where n is the number of unique tuples in the view. Hence, an insertion of a tuple involves reading of approximately

$$(u_0 d_i + u_1) \log n$$

bytes. Since $\binom{d}{d_i}$ attains a maximum around $d_i = \frac{1}{2}d$, the typical value of d_i is $d_i = \frac{1}{2}d$. Actual access to the tree nodes involves a number of pointer dereferences, such as looking for the left or right node and checking the node color. A comparison of attributes of a tuple stored in the node with attributes of the current tuple and creation of a new tree node takes $u_0 d$ memory accesses.

In addition, there is a number of auxilary operations, such as reading the smallest parent and updating pointer arrays. These operations involve $v_0 dn$ memory accesses. Hence, the total number of cycles required to compute a view is

$$\mu((u_0 d_i + u_1)n \log n + v_0 dn) + w_0,$$

where μ is the average number of machine cycles it takes to access a datum and w_0 is a constant number of bookkeeping operations incurred once per aii views.

The value of μ changes as the number of input tuples grows. If the L1 cache can hold a tree of depth l and $n \leq 2^l$, all tree node accesses are L1 hits and $\mu = M_1 + m_0$, where M_1 is the number of cycles it takes to access a datum in the L1 cache, and m_1 is time to access a tuple ammortized over all node accesses. For a two-level cache, if a tree of depth l fits in the main memory, and a cache of level i can hold a tree of depth l_i, the cost of insertion is

$$\mu = \frac{1}{l}(M_1 l_1 + M_2(l_2 - l_1) + M_0(l - l_2)),$$

[2] Specifically, if the machine has 64-bit pointers and the tuple attributes are 4 bytes long, the node size is $36 + 4 * d_i$.

where M_0 is the number of cycles to access a datum in main memory, and M_1 and M_2 are the number of cycles to access a datum in L1 and L2, respectively. Hence, as the number of input tuples grows, the tree spills out of L1, and then out of L2, the cost of memory access μ gradually increases.

If all tuples in all views are unique and the total number of generated tuples is $2^d n$, it takes $2^d \mu((u_0 d_i + u_1) n \log n + v_0 dn) + w_0$ cycles for accessing memory. After simplification and taking into account that a typical value of d_i is $d/2$, we conclude that the time per tuple is

$$T = \mu((v_1 d + u_1) \log n + v_0 d) + \frac{w_0}{n}, \tag{1}$$

where $v_1 = u_0/2$. TUPS of the algorithm equals T^{-1}.

This formula for time per tuple has a simple interpretation. For small n, the last term dominates the others, hence the time decreases as w_0/n. For large n the first term dominates the others, hence time per tuple is proportional to $\log n$, to the cost μ of access to the current level of memory, and to the number of attributes d. In practice, not all views have sizes n, and u_1 dominates $v_1 d$. As a result, equation (1) should be considered only an approximation.

4 Distributed Data Cube Computation

4.1 Parallel Data Cube Computation

There are a number of ways to perform DCO in parallel [4, 5, 19, 21]. The child-parent dependences among views usually are represented by a weighted lattice of the views. The weights of the nodes of the lattice (the view sizes) and of its edges (costs of calculating dependent views) are usually estimated. A common final step is a partitioning of a weighted spanning tree of the lattice into p balanced tasks, where p is the number of processors.

A method described in [5] creates a relatively small number of coarse grained independent tasks. First, it creates a spanning tree T of the view lattice with the view weights representing the cost of creating it from a matched parent in T. The partitioning of T into subtrees is an NP-complete problem [5]. So they use a heuristic approach, which creates the p balanced subproblems and minimizes the number of subtrees assigned to a processor. First, the min-max tree k-partitioning algorithm [2] is used to partition T into $s \cdot p$ subtrees, where $s \geq 1$ is an integer called *oversampling ratio*. Then, the partitioning uses a packing heuristic to assign s subtrees to the processors. The performance results [5] show that a partitioning with s equal 2 or 3 provides good load balancing across the processors.

In our parallel implementation of DCO computation we use a simpler algorithm. We use a priori knowledge of the view sizes to partition the data cube such that output data files (files with generated views) are well balanced across the processors. We distribute the output data across all view files evenly because the data cube computation is I/O bound. Since the size of the output data cube is usually significantly bigger than the size of input data, this approach yields a relatively good load balance. We partition the data cube into coarse grained independent tasks with little inter-task communication, so that the tasks can be executed on shared memory machines, clusters of shared memory machines, and in a distributed grid environment.

Assuming that the number of the processors p is substantially smaller than the number of the views, we assign the views to the processors in three steps. First, we sort all views by decreasing size. Then we assign the views to processors by *zig-zag folding*: j^{th}-view is assigned to processor $(-1)^e j + e(p-1) \mod p$, where $e = (j/p) \mod 2$. Finally, we restore child-parent relationships in the lists of views assigned to each processor by sorting the lists by the number of dimension attributes. Now each processor computes a view from the smallest parent. This approach gives a load balance exceeding 94% in our experiments.

4.2 Grid Level of Parallelization

Our experiments with the DC benchmark (see Figure 5) show that, on a parallel machine, we can use only a few processors efficiently. The reason is that the benchmark saturates the machine I/O devices. On the other hand, since there is little communication between DC tasks, the DC turn-around time can be reduced if some tasks will be executed on other grid machines.

When forming tasks for heterogeneous computational grids, we have to take into account that the i-th machine has a performance of τ_i TUPS. To achieve a good load balance, we have to assign to the i-th machine a load of $V * \tau_i/T$, where V is the total load and

$$T = \sum_{i=1}^{s} \tau_i.$$

To do that we use a modification of the zig-zag folding of Section 4.1. We sort all views by decreasing the sizes. Then we assign the views to the machines by scanning the machines alternately in directions of increasing and decreasing of i, skipping the machines whose load exceeds $V * \tau_i/T$. We then partition the tasks assigned to each machine, as described in Section 4.1.

5 DC Benchmark Results

We tested the DC Benchmark on the machines shown in Table 1 with normal production load during our experiments.

Table 1. The machines used in our experiments.

Machine Name	NP	Clock Rate (MHz)	Peak Perf. (GFLOPS)	Memory (GB)	Maker	Architecture	Batch System
SF880	8	900	14.4	16	SUN	UltraSparc 3	-
O3K1	1024	600	1200	256	SGI	Origin3800	PBS
O3K2	512	400	400	262	SGI	Origin3000	PBS
O3K3	256	400	200	98	SGI	Origin3000	PBS
U60/1	2	450	1.8	1	SUN	ULTRA60	-
U60/2	2	450	1.8	1	SUN	ULTRA60	-
O2K	32	250	16	8	SGI	Origin2000	-
G4	1	1250	2.5	1	Apple	G4	-
XEON	2	3060	-	2	Intel	Hyperthreaded	-

5.1 Single Processor Memory Signature

Exeprimental results of running a single processor version of the DC benchmark with 11, 12 and 14 dimensions are shown in Figures 1 and 2. Figure 1 clearly indicates presence of an initial segment and two straight line segments in each plot. The initial segment indicates domination of the last term of Equation 1. Each straight line segment indicates the $\log n$ term of Equation 1 with constant memory access cost μ. The end points of the segments reflect a change in the memory access cost μ when the *RB*-trees, used for sorting of the views, grow beyond the L1 and L2 caches. As a result, the minimum in the graphs indicates the point when the L1 cache is filled up by the tree, hence the size of the L1 cache can be estimated from the minimum and the tree node size ($4 * d + 36$, see Section 3.3).

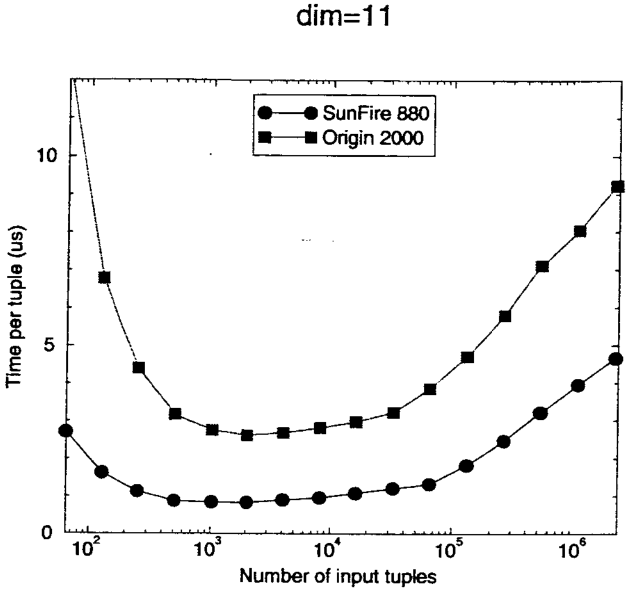

Figure 1. Time per tuple of DC with 11 dimensions on the SUNFire 880 and Origin 2000. Each curve consists of an initial segment, and two straight line segments. These segments are (64,512), (512,64K), and (64K,2M) for SUNFire 880 and (64,1K), (1K,32K), and (32K,2M) for Origin 2000, respectively.

Plots for dimensions 12 and 14 for three architectures are shown in Figure 2. These plots demonstrate that the structure (initial segment - two straight line segments) holds for other dimensions/architectures. This gives us us a base to call Time Per Tuple (TPT) of DC.U a memory signature.

5.2 Scalability of Memory Performance

The graphs of Figures 3 and 4 show that in-core computations of DC scale very well. The actual load imbalance was less than 6% for up to 32 processors. Comparison of the right graph on Figure 3 and graphs on Figure 4 shows that the overhead of memory initialization on Origin3800 is significantly larger than that on SUNFire 880, while for larger sizes of input data, memory performance of the two architectures is very close.[3]

5.3 Scalability of I/O Performance

For large data sets, the *RB*-trees do not fit in core memory, and Figure 5 shows that the increase in the number of processors does not improve TUPS (TUples Per Second). It indicates that the I/O system on the machine has been fully utilized. To avoid the I/O bottleneck, we distributed the DC across machines of a computational grid using the algorithm of Section 4.2. Such distribution incurs a small overhead, since, as soon as a parent view of a task is generated, the task does not have to communicate with others. The results shown in Table 2 demonstrate that TUPS increases when additional grid resources are used.

[3] The memory performance gap between the Origin 2000 and SUNFire 880 is about 2.5 and growing with the size of the input set, see Figure 1.

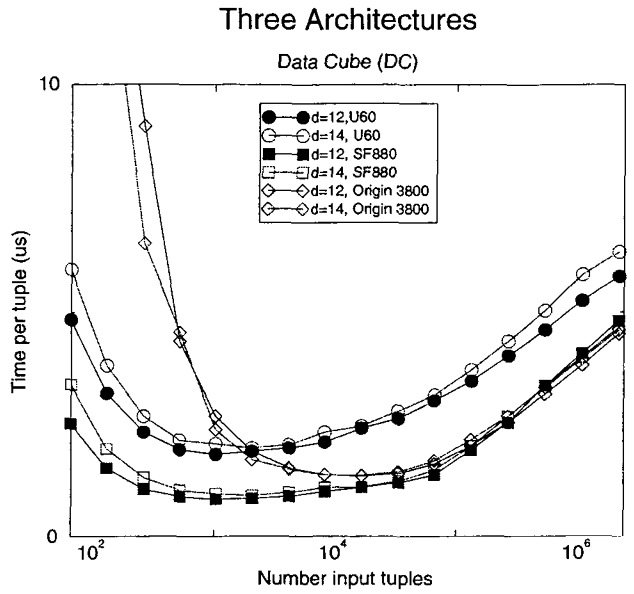

Figure 2. The Time per Tuple curves of DC 12 and 14 dimensions on the Ultra Sparc 60, SUNFire 880, and Origin 3800, respectively.

Table 2. Various load distributions between grid machines. The "np" indicates the number of processors used on the machine, the "load" is a fraction of the load per machine processor, hence "np"·"load" equals the fraction of work assigned to the machine.

Machine	Experiment Number											
	1		2		3		4		5		6	
	np	load	np	load	np	load	np	load	np	load	np	load
SF880	8	1/8	-	-	-	-	4	1/8	4	1/8	4	1/8
O3K1	-	-	8	1/8	-	-	4	1/8	8	1/32	-	-
O3K2	-	-	-	-	8	1/8	-	-	8	1/32	8	1/32
U60/1	-	-	-	-	-	-	-	-	-	-	2	1/16
U60/2	-	-	-	-	-	-	-	-	-	-	2	1/16
time (s)	242.2		179.1		179.3		106.2		93.1		93.1	
TUPS	$848 \cdot 10^3$		$1145.7 \cdot 10^3$		$1145.5 \cdot 10^3$		$1934.4 \cdot 10^3$		$2206.5 \cdot 10^3$		$2206.5 \cdot 10^3$	

5.4 Benchmark Classes

In practice, to use a significant number of DC instances as a benchmark would be confusing for the users. To get the memory signature for any particular machine, the user has to make runs of DC for different sizes of the input data sets. The verification values for a significant number of DC instances would constitute a large array of data to be distributed

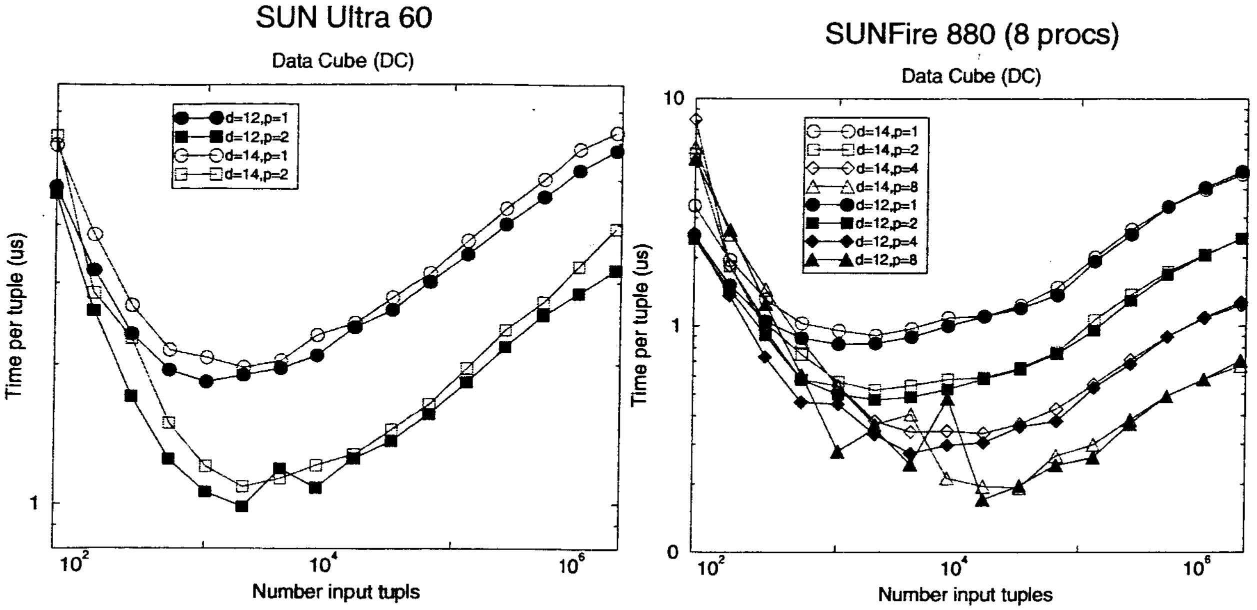

Figure 3. Scalability of Time per Tuple of DC on the Ultra Sparc 60, left pane, and on the SUNFire 880, right pane.

with the benchmark. To resolve these issues, we specify a few representative points in the ADC parameter space. This follows the NPB tradition to specify classes (S, W, A, B, C, and D) reflecting the computational effort required to perform the benchmark. In DC we define classes so that they will exercise all levels of memory hierarchy of current systems, from L1 to the I/O system. This restriction will not prevent a user from obtaining memory signatures using DC, but it will focus their experiments in the representative points in the ADC parameter space. For choosing classes, we use the flexibility provided by our choice of the data set generator.

For the benchmark we choose m_i to be prime numbers and g_i to be generators of $(\mathbb{Z}/m_i\mathbb{Z})^*$, hence having period $q_i = f_i = m_i - 1$. Also, we choose m_i such that $m_i - 1$ has many small prime factors so that $\mathrm{LCM}_{i \in I}(q_i)$ has a good chance of been small. This approach gives us full control over the sizes of the data set and its views. Our actual choice of the m_i is shown in Table 3.

We choose four groups of prime numbers $\{3, 5, 7\}$, $\{11, 13, 17, 19\}$, $\{23, 29, 31, 37\}$, and $\{41, 43, 47, 53, 59\}$. For each group we choose five smallest primes m_i such that prime factors of $m_i - 1$ are 2 and numbers from this group[4], Table 3. This set of parameters gives us a data set of $2^5{\cdot}3^2{\cdot}5^2{\cdot}7^2{\cdot}11{\cdot}13{\cdot}17{\cdot}19^2{\cdot}23{\cdot}29{\cdot}31^2{\cdot}37{\cdot}41{\cdot}43{\cdot}47{\cdot}53{\cdot}59$ different tuples and, for example, we can choose $n = 2 \cdot 11 \cdot 23 \cdot 41 \cdot 3 \cdot 13 \cdot 29 \cdot 43 \cdot 5 \cdot 17 = 85759918530$. At the same time, the sizes of five-dimensional views (relative to each of the groups) are small relative to the total number of the elements in the data set. We further restrict the set of parameters to make four classes of the benchmark: S, W, A, and B and reduce the sizes of the views having many attributes. For doing this, we designate subcubes generated by the first five, 10, 15, and 20 dimensions as classes S, W, A, and B respectively 3.

We also leave out parameters for the User defined class U. In this class, a user can specify any subset of the attributes and any number of tuples. For the class U, we do not provide checksums or verification values. The total number of tuples in each class, the sizes of input and estimated output files are shown in Table 4. The final results of the DC

[4] Since we use odd primes, $m_i - 1$ always has 2 as a factor.

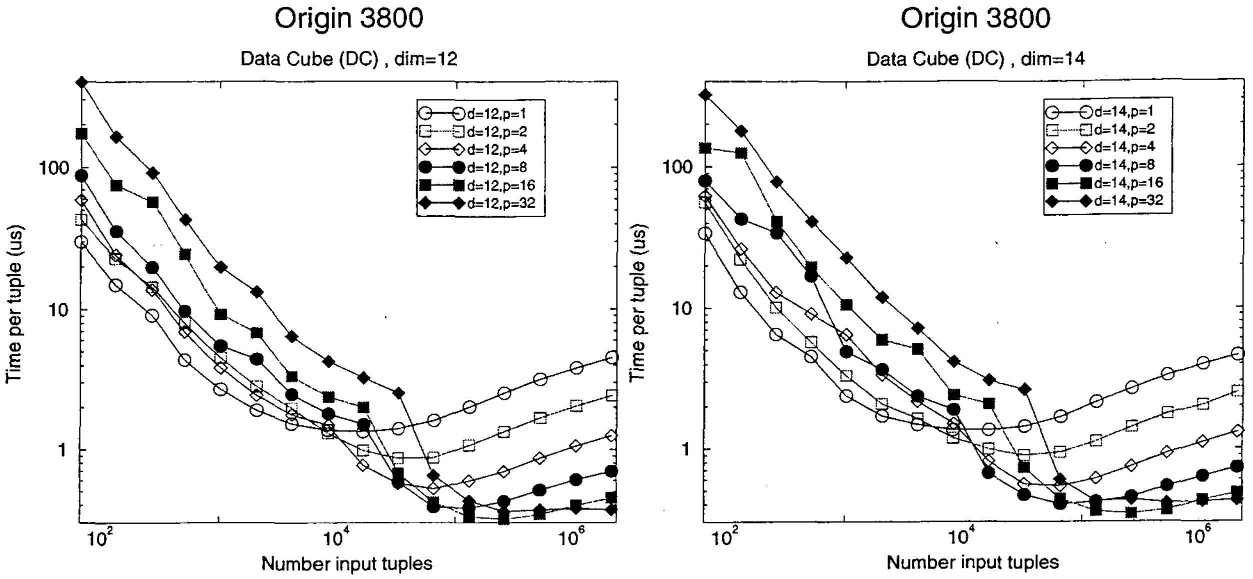

Figure 4. Scalability of Time per Tuple of DC.U on the Origin 3800.

benchmark on our experimental set of machines are shown in Table 5.

6 Related Work

The importance of memory performance in the overall assessment of system performance was recently acknowleged by the Innovative Computing Laboratory at The University of Tennessee at Knoxwille by adding three memory benchmarks *STREAM*, *PTRANS*, and *b_eff* to the LINPACK benchmark and creation of the HPC Challenge Benchmark suite [13]. The *STREAM* benchmark measures memory bandwidth by streaming very long vectors through the processor's registers and computing linear combinations of the vectors. The parallel matrix transpose *PTRANS* benchmark exercises communication capacity of the computer memory by transposing a large dense matrix. During the transpose, pairs of processors communicate with each other simultaneously. The *b_eff* (effective bandwidth) benchmark measures the effective bandwidth by simultaneously sending (MPI) messages using several communication patterns. The patterns are based on rings and on random distributions of the communicating processes.

The HINT benchmark [12] was used for probing sizes of the primary and secondary caches. Recently it was realized that similar probing of the memory caches can be accomplished by traversing the memory with a fixed stride [11, , p. 513]. Such a walk causes numerous cache and TLB misses and may result in low memory performance. This method is been used for fighting email spam by asking the sending computer to pay some computational cost per email message by solving a memory bound puzzle.

The benchmarking of data mining systems is a well established area of High Performance Computing [22, 28]. These benchmarks are designed to compare performance of query systems rather than to benchmark memory or I/O performance.

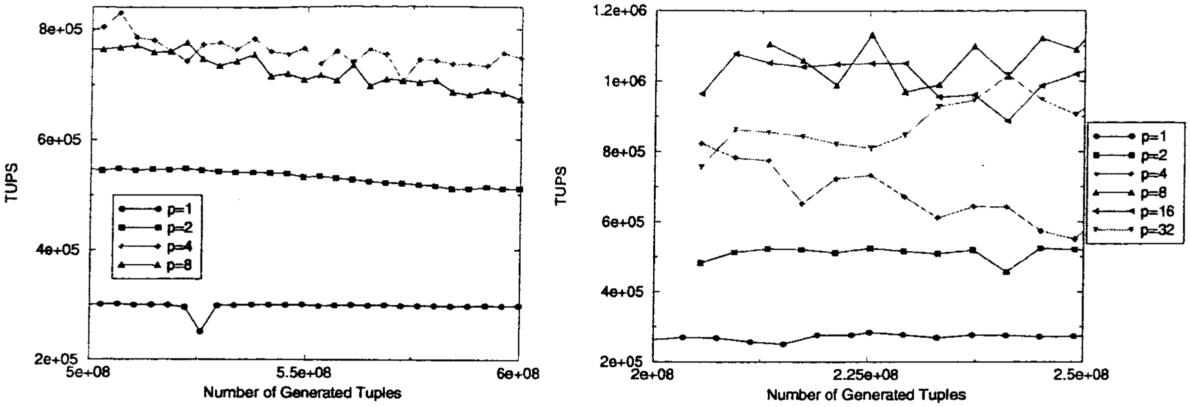

Figure 5. Performance of DC.U with 10 dimensions on the SUNFire 880 (left pane) and on the Origin3800 (right pane). There is no communication between the processes and the load balance exceeded 94%. In spite of an increase of the number of processors from 4 to 8 (for SUNFire 880) and from 8 to 32 (Origin 3800), the DC benchmark performance degrades due to a saturation of the I/O systems.

7 Summary

The DC benchmark represents an important set of computations used in OLAP and data mining. It executes $O(\log n)$ memory acesses per output tuple and is memory or I/O bound. The Arithmetic Data Sets used in DC are described by a small number of parameters and have a priori known sizes of the views. Parallelization of the DC incurs a small overhead and can be well balanced in load. We introduce the number of generated TUples Per Second (TUPS) as DC performance metric. The reciprocal of TUPS, Time Per Tuple gives a signature of the computer memory performance. We use charcteristic points in the signatures to choose parameters of various classes of DC. We provide a reference implementation of the DC benchmark and use it to benchmark TUPS of several computer architectures. We demonstrated that DC can saturate a machine's I/O system and in this case its performance can be improvemed by using additional grid resources.

Table 3. Dimensions of the Arithmetic Data Cube and generators for Classes S, W, A, and B. Here "Least Gen." γ_i is the smallest generator of $(\mathbb{Z}/m_i\mathbb{Z})^*$, and the "Exponent for the class" is e_i such that $g_i = \gamma_i^{e_i}$ for given class.

Prime	Factorization of $m-1$	Least Gen.	Exponent for the class					Seed
			U	S	W	A	B	
1. 421	$2^2 \cdot 3 \cdot 5 \cdot 7$	2	11	11	2^2	2^2	$2^2 \cdot 7$	211
2. 601	$2^3 \cdot 3 \cdot 5^2$	7	13	13	$2^3 \cdot 5$	$2^3 \cdot 5$	$2^3 \cdot 5$	301
3. 631	$2 \cdot 3^2 \cdot 5 \cdot 7$	3	17	17	$2 \cdot 3$	$2 \cdot 3$	$2 \cdot 3 \cdot 7$	316
4. 701	$2^2 \cdot 5^2 \cdot 7$	2	19	19	$2^2 \cdot 5$	$2^2 \cdot 5$	$2^2 \cdot 5 \cdot 7$	351
5. 883	$2 \cdot 3^2 \cdot 7^2$	2	23	23	$2 \cdot 3 \cdot 7$	$2 \cdot 3 \cdot 7$	$2 \cdot 3 \cdot 7^2$	442
6. 419	$2 \cdot 11 \cdot 19$	2	23		23	$2 \cdot 19$	$2 \cdot 19$	210
7. 443	$2 \cdot 13 \cdot 17$	2	29		29	$2 \cdot 13$	$2 \cdot 13$	222
8. 647	$2 \cdot 17 \cdot 19$	5	31		31	$2 \cdot 19$	$2 \cdot 19$	324
9. 21737	$2^3 \cdot 11 \cdot 13 \cdot 19$	31	37		2^2	$2^3 \cdot 13 \cdot 19$	$2^3 \cdot 13 \cdot 19$	10869
10. 31769	$2^3 \cdot 11 \cdot 19^2$	7	41		$2^2 \cdot 19$	$2^3 \cdot 19^2$	$2^3 \cdot 19^2$	15885
11. 1427	$2 \cdot 23 \cdot 31$	2	41			$2 \cdot 23$	$2 \cdot 31$	714
12. 18353	$2^4 \cdot 31 \cdot 37$	3	43			2^4	$2^4 \cdot 31$	9177
13. 22817	$2^5 \cdot 23 \cdot 31$	3	47			$2^5 \cdot 23$	$2^5 \cdot 31$	11409
14. 34337	$2^5 \cdot 29 \cdot 37$	3	53			2^5	$2^5 \cdot 29$	17169
15. 98717	$2^2 \cdot 23 \cdot 29 \cdot 37$	2	59			$2^2 \cdot 23$	$2^2 \cdot 29$	49359
16. 3527	$2 \cdot 41 \cdot 43$	5	3				$2 \cdot 43$	1764
17. 8693	$2^2 \cdot 41 \cdot 53$	3	5				2^2	4347
18. 9677	$2^2 \cdot 41 \cdot 59$	2	7				2^2	4839
19. 11093	$2^2 \cdot 47 \cdot 59$	2	11				$2^2 \cdot 47$	5547
20. 18233	$2^3 \cdot 43 \cdot 53$	3	13				$2^3 \cdot 43$	9117

Table 4. The main sizes of the ADC. The notation a:b:c in the Views Generated row indicates starting view:ending view:view number increment.

	U	S	W	A	B
Dimensions	-	5	10	15	20
Generator period	-	88200	9699690	653119005	306037160385
Number of tuples	-	10^3	10^5	10^6	10^7
ADC size	-	28 KB	4.8 MB	68 MB	880 MB
Views generated	all	all	all	0:2^{15} – 1:2^6	0:2^{20} – 1:2^{14}
Output size	-	547 KB	2.594 GB	17.84 GB	30.84 GB
Number of generated tuples	-	29232	89297411	454765673	595267023
Verification checksum	-	464620213	1401796434318	7141688178042	-

Table 5. Single processor DC performance. Classes S, W, and A were executed in-core, class B was executed out-of-core.

	CLASS							
	S		W		A		B	
Machine	TUPS	sec	TUPS	sec	TUPS	sec	TUPS	sec
SF880	1232897	0.02	624518	142.99	218774	2078.70	90575	6572.09
O3K1	175161	0.17	489962	182.25	234481	1939.46	114571	5196.62
O3K2	203182	0.14	345283	258.62	178296	2550.62	-	-
U60/1	549174	0.05	263525	338.86	155632	2922.06	-	-
O2K	176118	0.17	197893	451.24	105416	4314.01	-	-
G4	2019202	0.01	474797	188.07	-	-	-	-
XEON	2617712	0.01	766794	116.46	478907	949.59	-	-

Bibliography

[1] S. Agarwal, R. Agarwal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. *On the Computation of Multidimensional Aggregates.* Proceedings of the 22th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers, 1996, pp 506-521.

[2] R. Becker, S. Schach, and Y. Perl. *A Shifting Algorithm for Min-max Tree Partitioning.* J. ACM, (29), 58-67, 1982.

[3] K. S. Beyer and R. Ramakrishnan. *Bottom-up computation of sparse and iceberg cubes.* SIGMOD 1999, 359-370.

[4] Y. Chen, F. Dehne, T. Eavis, A. Rau-Chaplin. *Parallel ROLAP Data Cube Construction On Shared-Nothing Multiprocessors.*

[5] F. Dehne, T. Eavis, S. Hambrusch, A. Rau-Chaplin. *Parallelizing the Data Cube.* Kluwer Academic Publishers, 2001.

[6] M. Frumkin, R. F. Van der Wijngaart. *NAS Grid Benchmarks: A Tool for Grid Space Exploration.* Cluster Computing, Vol. 5, pp. 247-255, 2002.

[7] M. Frumkin, L. Shabanov. *Arithmetic Data Cube as a Data Intensive Benchmark.* NAS Technical Report, NAS-03-005. http://www.nas.nasa.gov/Research/Reports/techreports.html

[8] G. Graefe. *Query Evaluation Techniques for Large Databases.* ACM Computing Surveys, 25(2), 73-170, 1993

[9] J. Gray, A. Bosworth, A. Layman, and H. Prahesh. *Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total.* Microsoft Technical Report, MSR-TR-95-22, 1995.

[10] *Grid Benchmarking Research Group.* http://www.ggf.org/L_WG/wg.htm.

[11] J. L. Hennessy, D. A. Patterson. *Computer Architecture.* Third Edition, Morgan Kaufman Publishers, 2003.

[12] J.L. Gustafson, Q.O. Snell. *HINT: A new Way to Measure Computer Performance* http://hint.byu.edu

[13] *HPC Challenge Benchmark. http://icl.cs.utk.edu/hpcc/*

[14] J. Han, J. Pei, G. Dong, K. Wang. *Efficient Computation of Iceberg Cubes with Complex Measures.* SIGMOD'01, Santa Barbara, CA, May 2001, 1-12.

[15] V. Harinarayan, A. Rajaraman, and J. D. Ullman. *Implementing Data Cubes Efficiently.* In Proc. of ACM SIGMOD, pp. 205-216, Montreal, Canada, June 1996.

[16] *IBM Quest Synthetic Data Generation Code.* http://www.almaden.ibm.com/cs/quest/syndata.html.

[17] L.S.V. Lakshmanan, J. Pei, and J. Han. *Quotient Cube: How to Summarize the Semantics of a Data Cube.* In VLDB'02.

[18] W. Liang, M. E. Orlowska. *Computing Multidimensional Aggregates in Parallel.* International Conference on Parallel and Distributed Systems, Taiwan, 1998, 92-99.

[19] H. Lu, X. Huang, Z. Li. *Computing Data Cubes Using Massively Parallel Processors.*

[20] J.D. McCalpin. *STREAM Benchmark* http://www.cs.virginia.edu/stream/

[21] S. Muto, M. Kitsuregawa. *A Dynamic Load Balancing Strategy for Parallel Datacube Computation.* Proceedings of the second ACM international workshop on data warehousing and OLAP, 1999, 67-72.

[22] *OLAP Council / ABP-1 OLAP Benchmark, Release II,* http://www.olapcouncil.org.

[23] S. Sarawagi, R. Agrawal, and A. Gupta. *On computing the data cube.* Technical Report RJ10026, IBM Almaden Research Center, San Jose, CA, 1996.

[24] S. Sarawagi, R. Agrawal, and N. Megiddo. *Discovery-driven Exploration of OLAP Data Cubes.* In Proc. International Conf. of Extending Database Technology (EDBT'98), March 1998, 168-182.

[25] A. Snavely, G. Chun, H. Casanova, R. F. Van der Wijngaart, M. Frumkin. *Benchmarks for Grid Computing: A Review of Ongoing Efforts and Future Directions.* SIGMETRICS Performance Evaluation Review (PER), Vol. 30, No. 4, March 2003

[26] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. *Dwarf: Shrinking the PetaCube.* ACM SIGMOD international conference on management of data. Madison, Wisconsin, USA. 2002.

[27] A. Shukla, P. Deshpande, J.F. Naughton, and K. Ramasamy. *Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies* VLDB 1996, 522-531.

[28] *TPC BENCHMARK^{TM} D (Decision Support) Standard Specification,* Revision 1.3.1, http://www.tpc.org.

[29] W. Wang, J. Feng, H. Lu, and J. Xu Yu. *Condensed Cube: An Efficient Approach to Reducing Data Cube Size.* Proceedings of the 18th International Conference on Data Engineering, 2002, 155-165.

[30] R. F. Van der Wijngaart and P. Wong. *NAS Parallel Benchmarks I/O Version 2.4.* NAS Technical Report NAS-03-002, NASA Ames Research Center, Moffett Field, CA, 2003.

[31] Y. Zhao, P. M. Deshpande, and J.F. Naughton. *An Array-Based Algorithm for Simultaneous Multidimensional Aggregates.* Proc. of the 1997 ACM-SIGMOD Conf., 1997, 159-170.